process also takes time and consumes power. There can be situations in which the RAMset is cleaned to make room for new data, the RAMset is used for the new data, but returns back to the prior set local variables (saved to memory) relatively quickly. In fact, one can imagine a loop in the executable code in which a method invokes a new method each time through the loop. This repeated invocation of the new method may entail a clean operation and the exit from the new method back to the calling method will a corresponding flush and memory retrieval of the calling method's data. This repeated invocation of a called method and return back to the calling method (an "oscillation") at the boundary of the RAMset space (thereby forcing a clean, flush, etc.) can consume considerable power and time just cleaning the RAMset and then flushing and bringing the data back into the RAMset. The following embodiment solves this problem.

[0066] In accordance with a preferred embodiment, **FIG. 13** shows the RAMset **126** divided into two portions **680** and **682**. In some embodiments, the two portions may each represent one-half of the size of the RAMset, but in other embodiments the division between the two portions need not be equal. For purposes of this disclosure, the portion **680** is referred to as the "upper" portion (also referred to as portion "I") of the RAMset and portion **682** is the "lower" portion (portion "II"). Data (e.g., Java local variables) can be stored in either or both portions **680** and **682**. In accordance with the preferred embodiment, preferably only one portion at a time is actively used by the cache subsystem to store or retrieve data. The non-active portion may include valid data, or not, but, while inactive, is not used to store new data or provide data contained therein. The upper portion **680** can be the active portion at a given point in time, while lower portion **682** is thus inactive. Later, the lower portion **682** can become the active portion while the upper portion becomes active. Which portion is active can thus switch back and forth in accordance with the preferred embodiments and as illustrated in **FIG. 14** and discussed below.

[0067] The embodiment of the RAMset in multiple portions uses the commands listed in Table I.

TABLE I

COMMANDS

	Command	Description
1	SPP	Switch RAMset to scratch pad policy
2	CP	Switch RAMset to cache policy
3	UPPER CLEAN	Clean upper portion of RAMset to memory
4	LOWER CLEAN	Clean lower portion of RAMset to memory
5	UPPER FLUSH	Invalidate upper portion
6	LOWER FLUSH	Invalidate lower portion
7	R.SET(++)	Allocate new memory page and set RAMset base address accordingly in Full_Set_Tag register
8	R.SET(--)	Free current memory page and restore RAMset base address to previous base address in Full_Set_Tag register

The SPP and CP commands cause the RAMset to be in the SPP and CP modes as discussed previously. The UPPER CLEAN and LOWER CLEAN commands can be implemented using the D-RAMset-CleanRange and D-RAMset-CleanEntry commands to clean just the upper or lower portions, respectively. Similarly, the UPPER FLUSH and LOWER FLUSH commands can be implemented using the D-RAMset-FlushRange and D-RAMset-FlushEntry commands to flush just the upper or lower portions, respectively. The R.SET(++) command causes a new page of external memory **106** to be allocated and mapped to the RAMset using the base address of the new memory page. The previous base address of the RAMset is saved as part of the data in the RAMset. The R.SET(--) command essentially performs the reverse operation of the R.SET(++) command and frees the current external memory page while restoring the base address of the RAMset to the previous base address.

[0068] **FIG. 14** shows eight states of the RAMset. The eight states are identified with reference numerals **700**, **702**, **704**, **706**, **708**, **710**, **712**, and **714**. Each state of the RAMset shown illustrates the upper and lower portions discussed above with respect to **FIG. 13**. An "X" in one of the RAMset portions indicate that that particular portion is the active portion.

[0069] The RAMset may initialize into state **700**. In state **700**, the RAMset is in the SPP mode to permit the upper portion to be used to store data (e.g. local variables) but to avoid accesses to external memory **106** upon a cache miss. As explained above, a JAVA method typically requires an allocation of a portion of the RAMset for use for its local variables. Further, one method may invoke another method which, in turn, may invoke another method, and so on. Each such invoked method requires a new allocation of storage space in the RAMset. In state **700**, each such allocation falls within the upper portion which is the active portion.

[0070] At some point, however, an invocation of a new method may require an allocation of RAMset storage that may exceed the available unused capacity of the upper portion. At this point, the lower portion of the RAMset needs to be used to store additional local variables for the newly invoked method. The invocation of this new method is identified by arrow **701** which points to RAMset state **702**.

[0071] In RAMset state **702** (which is also in operated in the SPP mode), the lower portion of the RAMset is now the active portion. The lower portion therefore can be used to store local variables for the newly invoked method and any additional methods that are invoked therefrom. As explained above, each called method returns to its calling method. As such, the method that was invoked that caused the transition from the upper portion being active to the lower portion of the RAMset being active may eventually return to the calling method. The return to such method is illustrated with arrow **703**. Further, an oscillation may occur between such methods-the method that invoked a method causing the transition to the lower portion as well as the transition back from such method. This type of oscillation (identified by oppositely pointing arrows **701** and **703** in dashed circle **690**), however, is not as problematic as the oscillations noted above because the oscillation identified by arrows **701** and **703** do not require cleaning, flushing, or re-loading the RAMset. That is, no memory access is required to oscillate between the two RAMset states **700** and **702**. Because no memory accesses are required, such oscillations advantageously take less time and consume less power.

[0072] However, as more and more methods are invoked requiring allocations of the lower portion of the RAMset while in state **702**, eventually, the entire RAMset (i.e. both portions) may become full of valid data. At this point, any new method that is invoked will require an allocation of